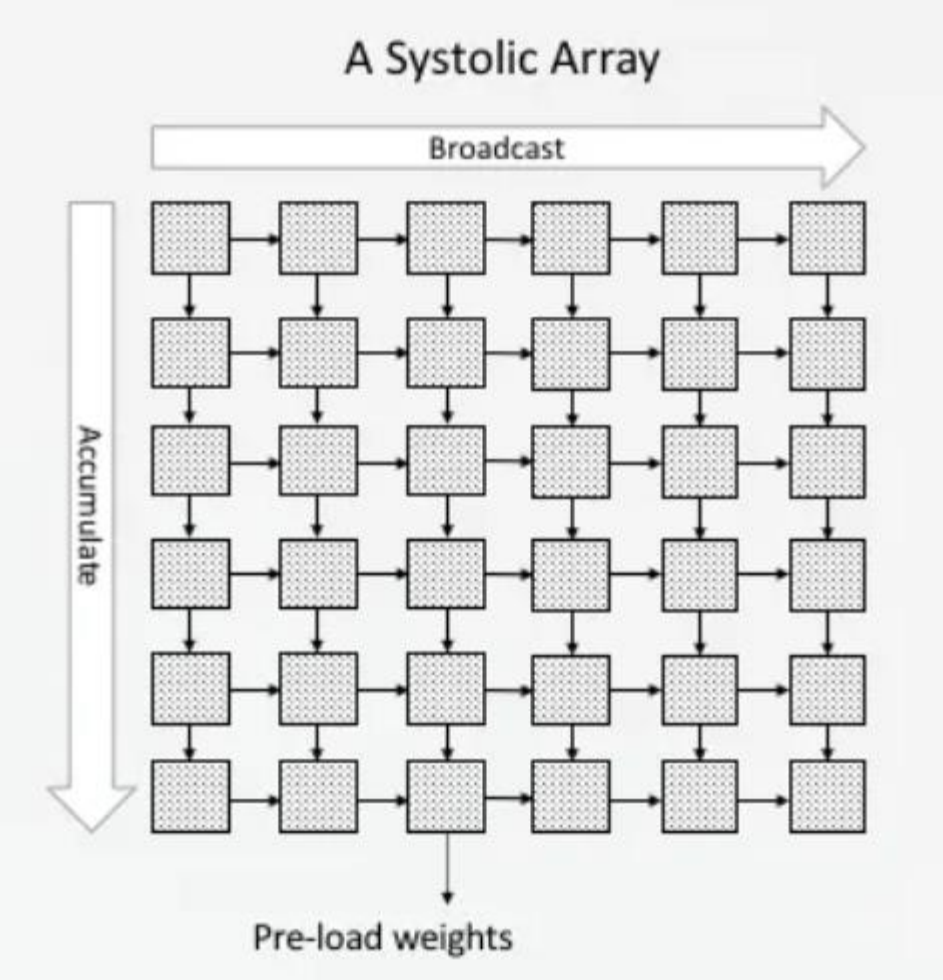
Systolic Array for Matrix Multiplication



Created by:

Mohamed Ahmed Mohamed Hussein 27/7/2025

Presented to:

Eng. Ahmed Abdelsalam

Systolic Array for Matrix Multiplication Overview

A **systolic array** is a hardware architecture composed of a grid of **Processing Elements (PEs)** that rhythmically compute and pass data through a fixed, regular flow. It is specially optimized for **parallel computation**, most notably for **matrix multiplication**, **convolution**, and other linear algebra operations.

In matrix multiplication, each PE is responsible for computing partial products and accumulating them to form the final result. The data (rows of one matrix and columns of the other) flows through the array **synchronously**, similar to how blood pulses through a heart — hence the name *systolic*.

Why Systolic Arrays Matter

- Traditional matrix multiplication is **computationally intensive** ($O(n^3)$ in naive form).
- Systolic arrays offer a **high-throughput, low-latency** hardware solution with:
    - **Data locality** (less memory movement)
    - **Pipelined computation** (parallelism)
    - **Scalability** (parameterizable grid size)

They are widely used in **AI hardware**, especially in **deep learning accelerators** like:

- Google's **TPU (Tensor Processing Unit)**
- Intel **Nervana**, NVIDIA **Tensor Cores**

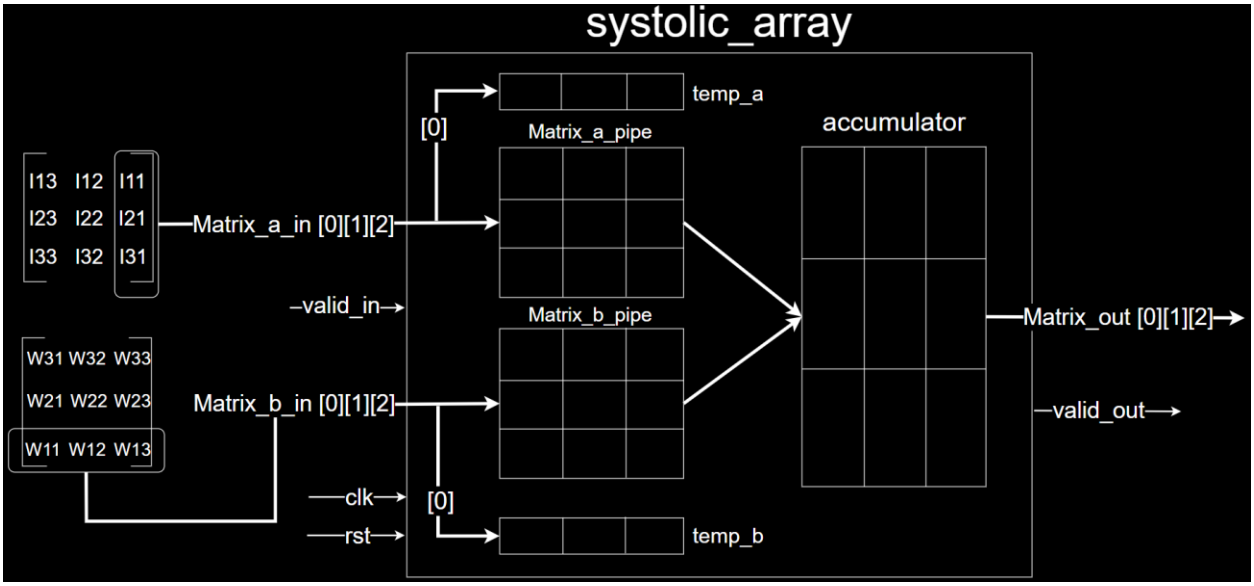
How Matrix Multiplication Works in a Systolic Array

In a systolic array:

- Matrix A is **fed column-by-column**, and its elements **move to the right**.
- Matrix B is **fed row-by-row**, and its elements **move downward**.
- Each PE receives one value from A and one from B each cycle, multiplies them, adds to an internal **accumulator**, and forwards the values to neighbors.

Architecture

Note: the architecture is made on the assumption the **input matrices are 3 by 3 matrices** just for illustration



Explanation

Notes:

- We will define each block functionality in the architecture
- We will go through a **clock by clock** visualization to explore how the design works
- We will explain how accumulator works
- We will have a **connect with code** section to ensure well connection between the explanation and the RTL code

1. Blocks definition

- **Temp_a:** to store the first element of every column of matrix A as we will need it later since it will be overwritten by the incoming inputs
- **Temp_b:** to store the first element of every row of matrix B as we will need it later since it will be overwritten by the incoming inputs
- **Matrix_a_pipe:** for pipelining the right inputs of matrix a every clock, better understanding in the clock by clock section
- **Matrix_b_pipe:** for pipelining the right inputs of matrix b every clock, better understanding in the clock by clock section

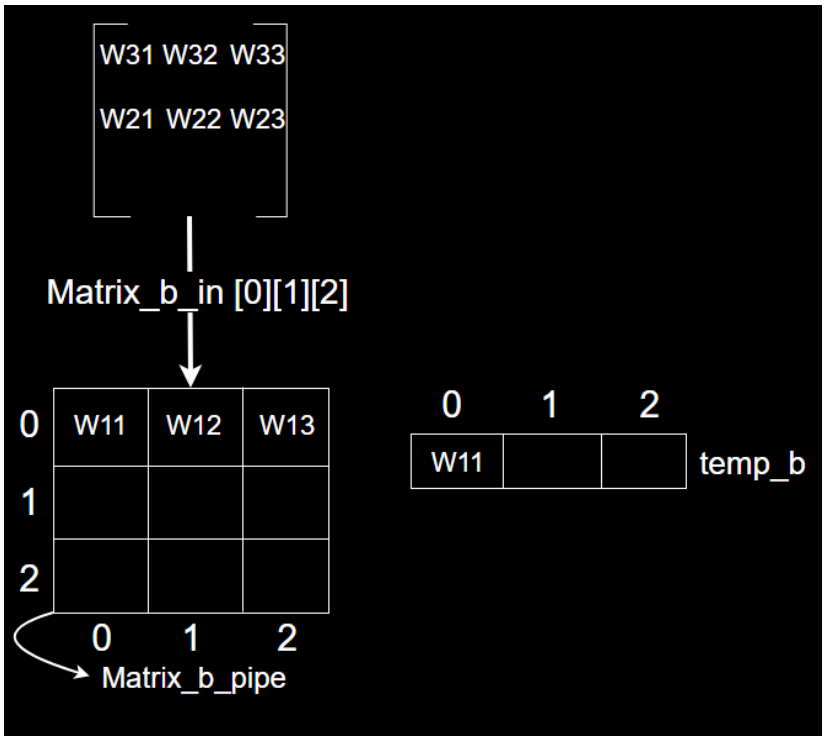
- **Accumulator:** to **sum the multiplication result** from matrix_a_pipe and matrix_b_pipe, its cells has the double width of either matrix_a_pipe or matrix_b_pipe

2. Clock by clock visualization

Note: we will have visualization **only for row wise feeding matrix**, by analogy it is the **same** for the other matrix but **column wise**

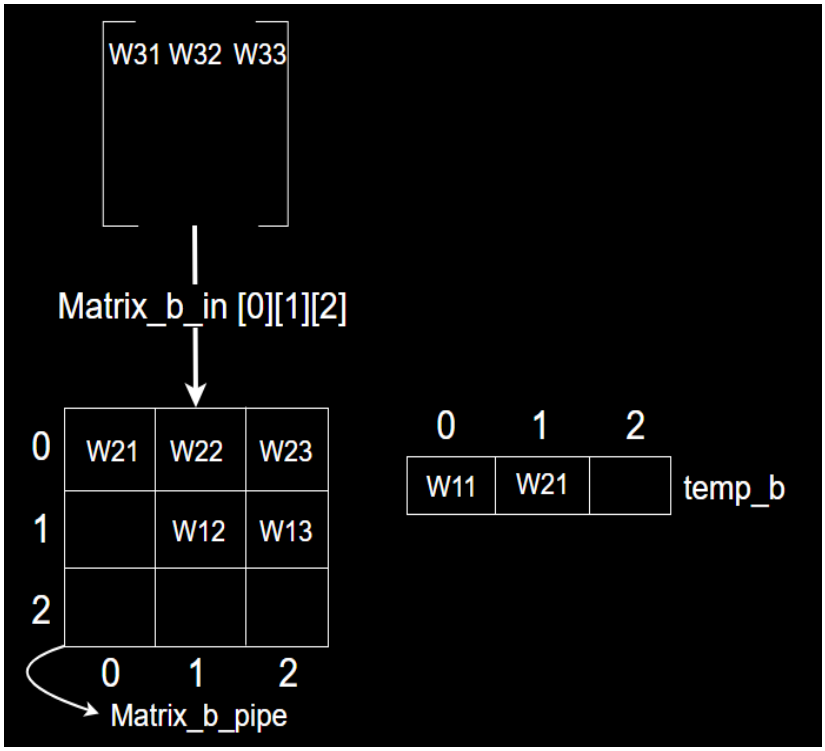
At 1st clock edge:

- We will take the example on the row feeding matrix (matrix b)
- At the first edge, the first input row (W11 W12 W13) is fed to the design through matrix_b_in
- The first row is stored in matrix_b_pipe aw we see
- The first element of the row (W11) is stored at temp_b

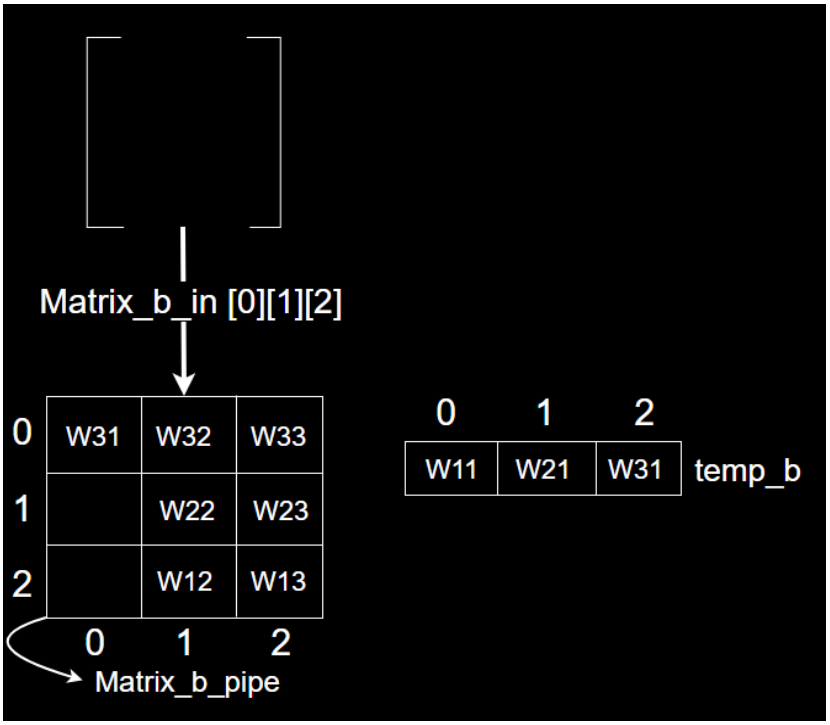


At 2nd clock edge:

- At the second edge, the second input row (W21 W22 W23) is fed to the design through matrix_b_in
- The second row is stored in matrix_b_pipe as we see
- The second element of the first row which is (W12) **is pipelined once as expected!**
- The third element of the first row which is (W13) is pipelined once and it **will be pipelined again at the following edge as the specs**
- The first element of the new row (W21) is stored at temp_b



At 3rd clock edge:



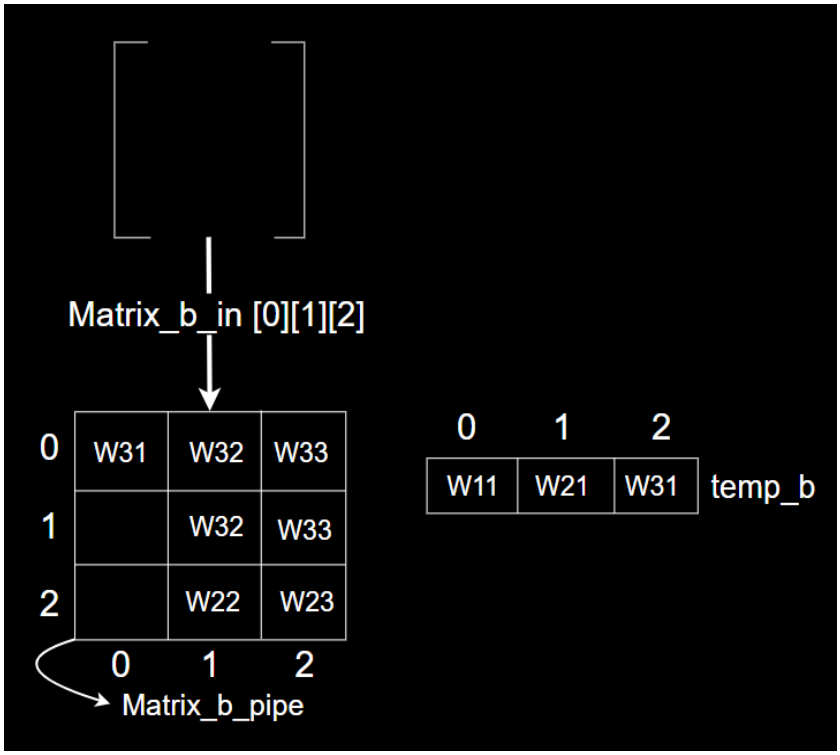
- At the third edge, the third row (W31 W32 W33) is fed to the design through matrix_b_in.
- The third row is stored in matrix_b_pipe as we see.
- The second element of the second row which is (W22) **is pipelined once as expected!**
- The third element of the second row which is (W23) is pipelined once and it **will be pipelined again at the following edge as the specs**
- The first element of the new row (W31) is stored at temp_b, **and temp_b now is locked and never accessed again until the next stimulus**
- The third element of the first row which was (W13) is **pipelined twice now as specs.**
- The second element of the the first row which was (W12) is **pipelined also for the second time**, but why?! Shouldn't it **be pipelined only once**?

  The answer is yes, but I only did this to save the value of it for later as when we evaluate later stages like O32, so we will need the value of (W12) as the equation is: **O32 = I31 * W12 + I32 * W22 + I33 * W32**, and it will be **overwritten** if I did otherwise, this approach will **not affect the functionality nor the timing.**

- **Here comes another question**, what about the **values of the first elements of each row input**, which have **no pipelining stages**, won't we need it too in later calculations?
  The answer is yes we will need them, **And here comes the role of temp_b!**
  In temp_b we **only store the first value of each input vector** to use them in later calculations and when we get the three values the **access to temp_b register is denied**.

At 4th clock edge:

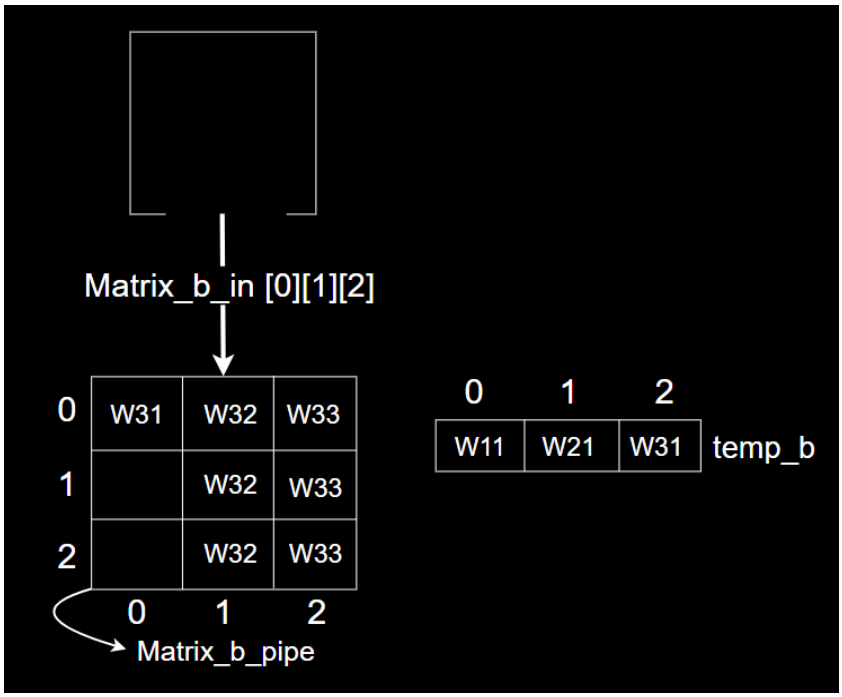
- The second element of the third row which is (W32) **is pipelined once as expected!**
- The third element of the third row which is (W33) is pipelined once and it **will be pipelined again at the following edge as the specs**
- The third element of the second row which was (W23) is **pipelined twice now as specs.**
- The second element of the the second row which was (W22) is **pipelined also for the second time, the reason is stated above**
- We see that temp_b register has nothing new as we expected



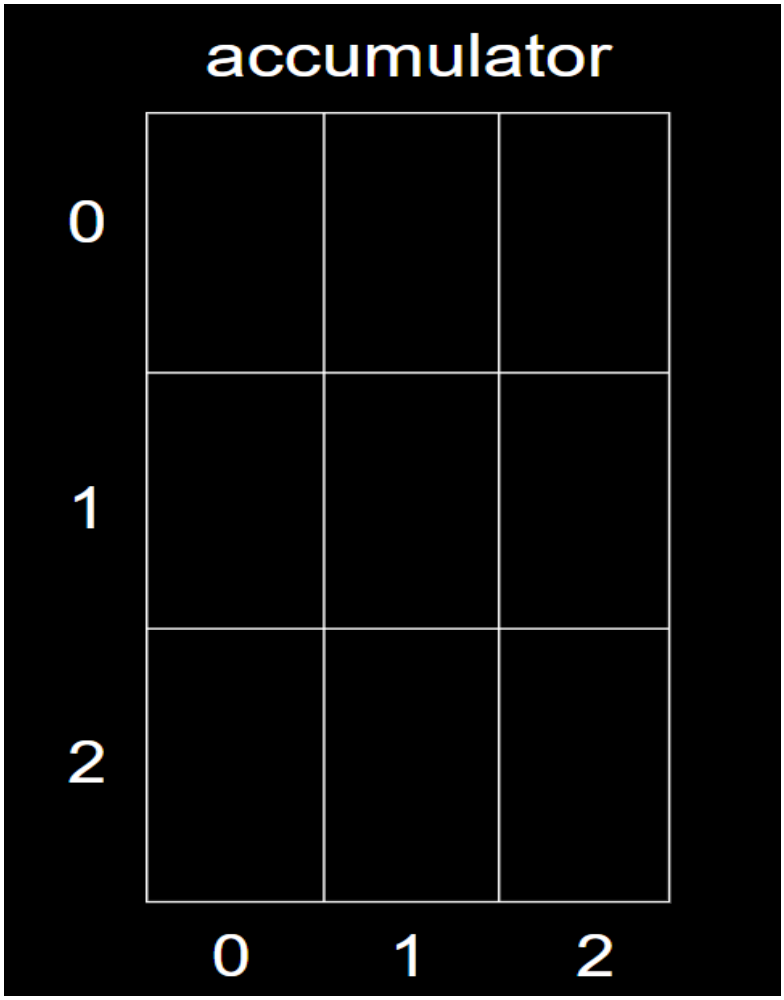
- Couldn't we **insert another input vectors** for other matrix to **benefit from the first row on matrix_b_pipe**?
  Yes we could but it is just **for simplicity I didn't**.

At 5th clock edge:

- The third element of the third row which was (W33) is **pipelined twice now as specs.**
- The second element of the the third row which was (W32) is **pipelined also for the second time, the reason is stated above**
- We see that temp_b register has nothing new as we expected



3. How accumulator works



- Each **cell** of the accumulator represents an **element** of the output matrix
- For example, the **accumulator[0][0] represents O[0][0]**
- So we compute the sum of the multiplication between matrix a and matrix b in the accumulator and then **assign it to the output matrix vector**
- For examble, if the accumulator [0][0], [0][1], [0][2] has the values 1, 3, 4 respectively, so the matrix_out vector will have the values 1, 3, 4 for the first row
- The first row of the accumulator is **computed faster than the following rows**, so we assign it to output vector **once it is done** to achieve **smallest delay possible**.
- Accumulator takes his inputs from matrix_a_pipe, matrix_b_pipe, temp_a and temp_b, he **calls the value he wants based on the equation of the cell (element) he will compute**

4. Connect with code

- Code has seven internals that are commented with their functionality in the code

```
reg [DATAWIDTH-1:0] matrix_a_pipe [0:N_SIZE-1][0:N_SIZE-1]; // to pipeline the inputs in second and third row from the column feeding matrix
reg [DATAWIDTH-1:0] matrix_b_pipe [0:N_SIZE-1][0:N_SIZE-1]; // to pipeline the inputs in second and third column from the row feeding matrix
reg [DATAWIDTH*2-1:0] accumulator [0:N_SIZE-1][0:N_SIZE-1]; // to store the multiplication results

reg [DATAWIDTH-1:0] temp_a [0:N_SIZE-1]; // to store the first row of the matrix A
reg [DATAWIDTH-1:0] temp_b [0:N_SIZE-1]; // to store the first column of the matrix B
reg [N_SIZE*2-1:0] clock_cycle; // to count the clock cycles for pipelining,
reg [N_SIZE*2-1:0] delayed_clock; // delayed clock is to be used in output logic as
//when we fill the pipe matrices and then the accumulator so it adds a one clock delay overhead for 3 by 3 matrices
```

- Code is splitted into **two always blocks** one is **sequential with the clock** and the other is **combinational**
- The **sequential always** block is used for **resetting, accumulation and pipelining** and it is also **splitted into five sections**
    - **Resetting section:**

```
// internal always block to handle the pipelining and accumulation
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        k <= 0; // for pipelining the inputs
        q <= 0; // for pipelining the inputs
        clock_cycle <= 0; // resetting the clock cycle counter
        for (i = 0; i < N_SIZE; i = i + 1) begin // output is zeroed on reset
            for (j = 0; j < N_SIZE; j = j + 1) begin
                matrix_a_pipe[i][j] <= 0; // zeroing the pipelined inputs
                matrix_b_pipe[i][j] <= 0; // zeroing the pipelined inputs
                accumulator[i][j] <= 0; // zeroing the accumulator
            end
        end
    end
```

    - **Storing section:**

```
else if (valid_in || k < N_SIZE*2) begin
    // storing the first elements of of both matrices in temp_a and temp_b
    if (clock_cycle < N_SIZE) begin
        for (q = 0; q < N_SIZE; q = q + 1) begin
            matrix_a_pipe[q][0] <= matrix_a_in[q]; // storing the column elements of matrix A
        end
        for (q = 0; q < N_SIZE; q = q + 1) begin
            matrix_b_pipe[0][q] <= matrix_b_in[q]; // storing the row elements of matrix B
        end
        temp_a[k] <= matrix_a_in[0]; // storing the first element of every column of matrix A as we will need it later
        // and they will be overwritten by the incoming inputs
        temp_b[k] <= matrix_b_in[0]; // storing the first element of every row of matrix B as we will need it later
        accumulator[0][0] <= accumulator[0][0] + (matrix_a_in[0] * matrix_b_in[0]); // ex: O11 = I11 * W11 + I12 * W21 + I13 * W31,
        // get the first row column multiplication result
    end
```

- **Pipelining section:**

```
// next stages are for pipelining
if (k > 0) begin
    for (q = 1; q < N_SIZE; q = q + 1) begin // this for loop is to pipeline the inputs for the right number of times
        matrix_a_pipe[q][k] <= matrix_a_pipe[q][k-1]; // for ex if N_size is 3 so the first element of the second row will be pipelined once,
    end // and the first element of the third row will be pipelined twice
    for (q = 1; q < N_SIZE; q = q + 1) begin
        matrix_b_pipe[k][q] <= matrix_b_pipe[k-1][q]; // pipelining the inputs in next columns
    end
end
if (k > 1 && N_SIZE > 1) begin // the above if statement is used for the first set of inputs from both matrices, so we have to put this block N_SIZE times
    for (q = 1; q < N_SIZE; q = q + 1) begin // this for loop is to pipeline the inputs for the right number of times
        matrix_a_pipe[q][k-1] <= matrix_a_pipe[q][k-2]; // for ex if N_size is 3 so the first element of the second row will be pipelined once,
    end // and the first element of the third row will be pipelined twice
    for (q = 1; q < N_SIZE; q = q + 1) begin
        matrix_b_pipe[k-1][q] <= matrix_b_pipe[k-2][q]; // pipelining the inputs in next columns
    end
end
if (k > 2 && N_SIZE > 2) begin
    for (q = 1; q < N_SIZE; q = q + 1) begin
        matrix_a_pipe[q][k-2] <= matrix_a_pipe[q][k-3];
    end
    for (q = 1; q < N_SIZE; q = q + 1) begin
        matrix_b_pipe[k-2][q] <= matrix_b_pipe[k-3][q]; // pipelining the inputs in next columns
    end
end
if (k > 3 && N_SIZE > 3) begin
    for (q = 1; q < N_SIZE; q = q + 1) begin
        matrix_a_pipe[q][k-3] <= matrix_a_pipe[q][k-4];
    end
    for (q = 1; q < N_SIZE; q = q + 1) begin
        matrix_b_pipe[k-3][q] <= matrix_b_pipe[k-4][q];
    end
end
```

- **Accumulation section:**

```
  // storing the multiplication result in the accumulator
  if (clock_cycle > 1 && (clock_cycle < N_SIZE + 2) && N_SIZE > 1) begin
      accumulator[0][1] <= accumulator[0][1] + (temp_a[k-2] * matrix_b_pipe[1][1]); // ex: O12 = I11 * W12 + I12 * W22 + I13 * W32
      accumulator[1][0] <= accumulator[1][0] + (temp_b[k-2] * matrix_a_pipe[1][1]); // ex: O21 = I21 * W11 + I22 * W21 + I23 * W31
      accumulator[1][1] <= accumulator[1][1] + (matrix_a_pipe[1][1] * matrix_b_pipe[1][1]); // ex: O22 = I21 * W12 + I22 * W22 + I23 * W32,
      // as I21 will be I22 in the next cycle and I22 will be I23 in the next cycle
  end
  if (clock_cycle > 2 && (clock_cycle < N_SIZE + 3) && N_SIZE > 2) begin // if N_SIZE is greater than 2, we can perform the next multiplication
      accumulator[0][2] <= accumulator[0][2] + (temp_a[k-3] * matrix_b_pipe[2][2]); // ex: O13 = I31 * W11, I32 * W21, I33 * W31
      accumulator[1][2] <= accumulator[1][2] + (matrix_a_pipe[1][2] * matrix_b_pipe[2][2]); // ex: O23 = I21 * W13, I22 * W23, I23 * W33
      accumulator[2][0] <= accumulator[2][0] + (temp_b[k-3] * matrix_a_pipe[2][2]); // ex: O31 = I31 * W11, I32 * W21, I33 * W31
      accumulator[2][1] <= accumulator[2][1] + (matrix_b_pipe[2][1] * matrix_a_pipe[2][2]); // ex: O32 = I31 * W12, I32 * W22, I33 * W32
      accumulator[2][2] <= accumulator[2][2] + (matrix_a_pipe[2][2] * matrix_b_pipe[2][2]); // ex: O33 = I31 * W13, I32 * W23, I33 * W33
  end
  if (clock_cycle > 3 && (clock_cycle < N_SIZE + 4) && N_SIZE > 3) begin // if N_SIZE is greater than 3, we can perform the next multiplication
      accumulator[0][3] <= accumulator[0][3] + (temp_a[k-4] * matrix_b_pipe[3][3]);
      accumulator[1][3] <= accumulator[1][3] + (matrix_a_pipe[1][3] * matrix_b_pipe[3][3]);
      accumulator[2][3] <= accumulator[2][3] + (matrix_a_pipe[2][3] * matrix_b_pipe[3][3]);
      accumulator[3][0] <= accumulator[3][0] + (temp_b[k-4] * matrix_a_pipe[3][3]);
      accumulator[3][1] <= accumulator[3][1] + (matrix_b_pipe[3][1] * matrix_a_pipe[3][3]);
      accumulator[3][2] <= accumulator[3][2] + (matrix_b_pipe[3][2] * matrix_a_pipe[3][3]);
      accumulator[3][3] <= accumulator[3][3] + (matrix_a_pipe[3][3] * matrix_b_pipe[3][3]);
  end
```

- **Counters section:**

```
        // counters
        k <= k + 1; // incrementing the counter for pipelining
        clock_cycle <= clock_cycle + 1; // incrementing the clock cycle counter
        delayed_clock <= clock_cycle; // storing the current clock cycle in the delayed clock register
    end
    else
        delayed_clock <= clock_cycle; // to make the last clock
end
```

- **Combinational always** which is the **output stage** of the design and it is splitted into **two sections**
    - **Resetting section:**

```
// combinatoinal always block to output the results
always @(*) begin
    if (!rst_n) begin
        valid_out = 0;
        for (i = 0; i < N_SIZE; i = i + 1) begin
            matrix_out[i] = 0; // zeroing the output on reset
        end
    end
```

    - **Output section:**
        - **If 2 x 2 matrix:**

```
else if (clock_cycle == 3  && N_SIZE < 3) begin // the second condition to prevent entering this branch if N_SIZE is greater than 2
    for (i = 0; i < N_SIZE; i = i + 1) begin
        matrix_out[i] = accumulator[0][i]; // output the first row
    end
    if (N_SIZE == 2) matrix_out[1] = accumulator[0][1] + (temp_a[1] * matrix_b_pipe[1][1]); // just to fasten the assignment of the last pipelined element
    valid_out = 1; // setting the valid output flag
end
else if ((delayed_clock == 4 && N_SIZE < 4) || (N_SIZE == 2 && clock_cycle == 4)) begin // the last condition is specially for 2 x 2 matrices to make it generate the output in 4 clock cycles (perfect)
    for (i = 0; i < N_SIZE; i = i + 1) begin
        if (N_SIZE < 3)
            matrix_out[i] = accumulator[1][i]; // output of the second row if N_SIZE < 3
        else
            matrix_out[i] = accumulator[0][i]; // output the first row
    end
    if (N_SIZE == 3) matrix_out[2] = accumulator[0][2] + (temp_a[2] * matrix_b_pipe[2][2]); // just to fasten the assignment of the last pipelined element
    valid_out = 1; // setting the valid output flag
end
```

        - **If 3 x 3 matrix:**

```
else if ((delayed_clock == 4 && N_SIZE < 4) || (N_SIZE == 2 && clock_cycle == 4)) begin // the last condition is specially for 2 x 2 matrices to make it generate the output in 4 clock cycles (perfect)
    for (i = 0; i < N_SIZE; i = i + 1) begin
        if (N_SIZE < 3)
            matrix_out[i] = accumulator[1][i]; // output of the second row if N_SIZE < 3
        else
            matrix_out[i] = accumulator[0][i]; // output the first row
    end
    if (N_SIZE == 3) matrix_out[2] = accumulator[0][2] + (temp_a[2] * matrix_b_pipe[2][2]); // just to fasten the assignment of the last pipelined element
    valid_out = 1; // setting the valid output flag
end
else if (delayed_clock == 5 && N_SIZE < 5) begin // the second condition to prevent entering this branch if N_SIZE
    for (i = 0; i < N_SIZE; i = i + 1) begin
        if (N_SIZE < 4)
            matrix_out[i] = accumulator[1][i]; // output the second row
        else
            matrix_out[i] = accumulator[0][i]; // output the first row
    end
    if (N_SIZE == 4) matrix_out[3] = accumulator[0][3] + (temp_a[3] * matrix_b_pipe[3][3]); // just to fasten the assignment of the last pipelined element
    valid_out = 1; // setting the valid output flag
end
else if (delayed_clock == 6 && N_SIZE < 6) begin // the second condition to prevent entering this branch if N_SIZE
    for (i = 0; i < N_SIZE; i = i + 1) begin
        if (N_SIZE < 4)
            matrix_out[i] = accumulator[2][i]; // output the third row
        else
            matrix_out[i] = accumulator[1][i];
    end
    valid_out = 1; // setting the valid output flag
end
```

- **If 4 x 4 matrix:**

```
        else if (delayed_clock == 5 && N_SIZE < 5) begin // the second condition to prevent entering this branch if N_SIZE
            for (i = 0; i < N_SIZE; i = i + 1) begin
                if (N_SIZE < 4)
                    matrix_out[i] = accumulator[1][i]; // output the second row
                else
                    matrix_out[i] = accumulator[0][i]; // output the first row
            end
            if (N_SIZE == 4) matrix_out[3] = accumulator[0][3] + (temp_a[3] * matrix_b_pipe[3][3]); // just to fasten the assignment of the last pipelined element
            valid_out = 1; // setting the valid output flag
        end
        else if (delayed_clock == 6 && N_SIZE < 6) begin // the second condition to prevent entering this branch if N_SIZE
            for (i = 0; i < N_SIZE; i = i + 1) begin
                if (N_SIZE < 4)
                    matrix_out[i] = accumulator[2][i]; // output the third row
                else
                    matrix_out[i] = accumulator[1][i];
            end
            valid_out = 1; // setting the valid output flag
        end
        else if (delayed_clock == 7 && N_SIZE < 7) begin
            for (i = 0; i < N_SIZE; i = i + 1) begin
                matrix_out[i] = accumulator[2][i]; // output the third row
            end
            valid_out = 1; // setting the valid output flag
        end
        else if (delayed_clock == 8 && N_SIZE < 8) begin // the second condition to prevent entering this branch if
            for (i = 0; i < N_SIZE; i = i + 1) begin
                matrix_out[i] = accumulator[3][i]; // output the fourth row
            end
        end
        else begin
            valid_out = 0;
            for (i = 0; i < N_SIZE; i = i + 1) begin
                matrix_out[i] = 0; // zeroing the output
            end
        end
    end
endmodule
```

- Code is **well commented** and here is a **directed link to the code**: systolic_array.sv
- We can deduce that the code is **capable of performing matrix multiplication between 2 x 2, 3 x 3 and 4 x 4 matrices!** And we will have a test example for each one of them in **verifying functionality** section.
- I could've done more like 5 x 5 or 6 x 6 by inserting more of these code blocks:

```
if (k > 3 && N_SIZE > 3) begin
    for (q = 1; q < N_SIZE; q = q + 1) begin
        matrix_a_pipe[q][k-3] <= matrix_a_pipe[q][k-4];
    end
    for (q = 1; q < N_SIZE; q = q + 1) begin
        matrix_b_pipe[k-3][q] <= matrix_b_pipe[k-4][q];
    end
end
```

but I found it would be **impractical**, there must be an easier way to parametrize the design

```
if (clock_cycle > 3 && (clock_cycle < N_SIZE + 4) && N_SIZE > 3) begin // if N_SIZE is greater than 3, we can perform the next multiplication
    accumulator[0][3] <= accumulator[0][3] + (temp_a[k-4] * matrix_b_pipe[3][3]);
    accumulator[1][3] <= accumulator[1][3] + (matrix_a_pipe[1][3] * matrix_b_pipe[3][3]);
    accumulator[2][3] <= accumulator[2][3] + (matrix_a_pipe[2][3] * matrix_b_pipe[3][3]);
    accumulator[3][0] <= accumulator[3][0] + (temp_b[k-4] * matrix_a_pipe[3][3]);
    accumulator[3][1] <= accumulator[3][1] + (matrix_b_pipe[3][1] * matrix_a_pipe[3][3]);
    accumulator[3][2] <= accumulator[3][2] + (matrix_b_pipe[3][2] * matrix_a_pipe[3][3]);
    accumulator[3][3] <= accumulator[3][3] + (matrix_a_pipe[3][3] * matrix_b_pipe[3][3]);
end
```

Verifying Functionality

Notes:

- In this section we will verify the **both functionality and timing** of each test case
- Our test cases are: **2 x 2 matrix 2 x 2, 3 x 3 and 4 x 4 matrices.**
- The clock and clock counter will be seen in **cyan** for better visualization
- Output will be in **magenta** for better visualization
- If you want to test the 2 x 2 example, you should uncomment its stimulus in the testbench code and modify N_SIZE signal to be 2, do the same with the other cases
- I used **ChatGPT** to solve matrices

1. 2 x 2 input matrix test:

Input matrix a:

$$\begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} = \begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix}$$

Input matrix b:

$$\begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix} = \begin{vmatrix} 5 & 6 \\ 7 & 8 \end{vmatrix}$$

Expected output:

$C_{11} = 1 \times 5 + 2 \times 7 = 19$

$C_{12} = 1 \times 6 + 2 \times 8 = 22$

$C_{21} = 3 \times 5 + 4 \times 7 = 43$

$C_{22} = 3 \times 6 + 4 \times 8 = 50$

- **Feeding first vector:**

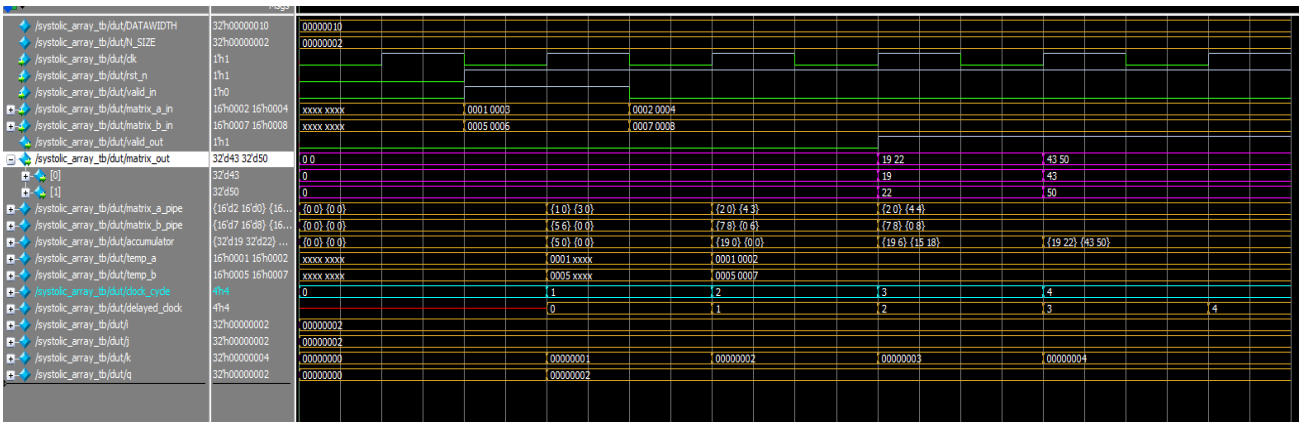
```
// -------------------------------
// First Input Cycle
// A feeds from left (column-wise), B feeds from top (row-wise)
//
//  Matrix A column:        Matrix B row:
//      →                      ↓
//    [ 1 ]                 [ 5  6 ]
//    [ 3 ]
// -------------------------------
valid_in = 1;
matrix_a_in[0] = 1;  // A11
matrix_a_in[1] = 3;  // A21
matrix_b_in[0] = 5;  // B11
matrix_b_in[1] = 6;  // B12
```

- **Feeding second vector:**

```
// -------------------------------
// Second Input Cycle
// A feeds second column, B feeds second row
//
//  Matrix A column:         Matrix B row:
//      →                       ↓
//    [ 2 ]                  [ 7  8 ]
//    [ 4 ]
// -------------------------------
#20; // negedge
matrix_a_in[0] = 2;  // A12
matrix_a_in[1] = 4;  // A22
matrix_b_in[0] = 7;  // B21
matrix_b_in[1] = 8;  // B22

// Stop feeding
valid_in = 0;
```

Waveform generated



So as we expected the **first output** vector at the **third clock** is **correct** and the output of the **second vector** at the **fourth clock** is also **correct!**

Transcript

```
# output matrix rows are 19,  22
# output matrix rows are 43,  50
# ====================================
# Expected Output Matrix C:
# C[1][1] = 19    C[1][2] = 22
# C[2][1] = 43    C[2][2] = 50
# ====================================
```

There is a monitor that monitors the output vector

Summary:

- **Functionality: Correct**
- **Timing**: perfect!

2. 3 x 3 input matrix test:

Input matrix a:

1	2	3
4	5	6
7	8	9

Input matrix b:

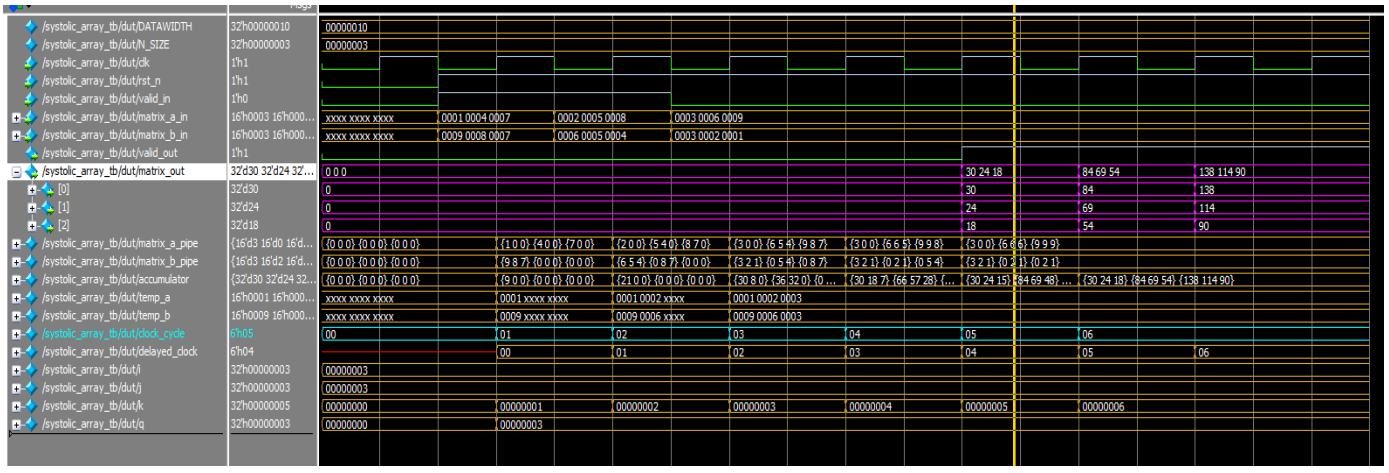
9	8	7
6	5	4
3	2	1

Expected output:

Row	C_1	C_2	C_3
1	30	24	18
2	84	69	54
3	138	114	90

Note: you will find also the visualization for the two matrix like above in the systolic_array_tb.sv file

Waveform generated



So as we expected the **first output** vector at the **fifth clock** is **correct**, the output of the **second vector** at the **sixth clock** is **correct and the third vector** at the **seventh clock** is also **correct,** but we notice that the first output is **delayed by one clock** cycle as it **should've been ready at the fourth clock not the fifth, so that's why I used the delayed clock**

Transcript

```
# output matrix rows are 30,  24,  18
# output matrix rows are 84,  69,  54
# output matrix rows are 138,  114,  90
# ====================================
# Expected Output Matrix C:
# C[1][1] = 30   C[1][2] = 24   C[1][3] = 18
# C[2][1] = 84   C[2][2] = 69   C[2][3] = 54
# C[3][1] = 138  C[3][2] = 114  C[3][3] = 90
# ====================================
```

Summary:

- **Functionality: Correct**
- **Timing**: delayed by one clock cycle

3. 4 x 4 input matrix test:

Input matrix a:

```
| 1   2   3   4  |
| 5   6   7   8  |
| 9  10  11  12 |
|13  14  15  16 |
```

Input matrix b:

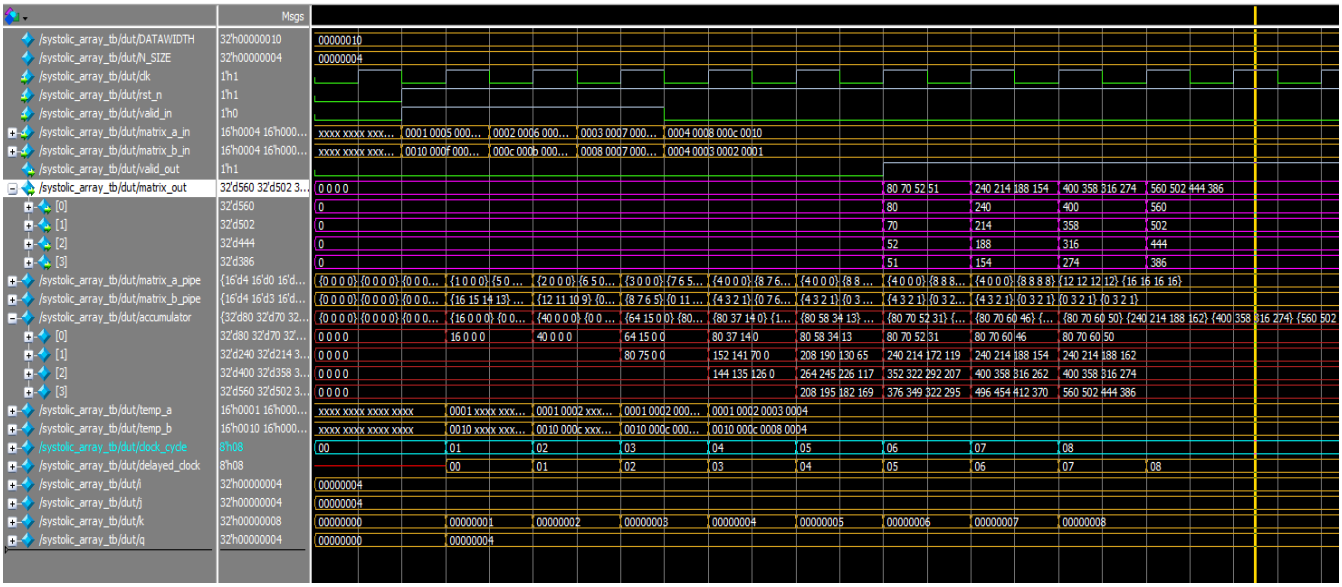
```
|16  15  14  13|
|12  11  10  9 |
| 8   7   6  5 |
| 4   3   2  1 |
```

Expected output:

	C_1	C_2	C_3	C_4
1	80	70	60	50
2	240	214	188	162
3	400	358	316	274
4	560	502	444	386

Note: you will find also the visualization for the two matrix like above in the systolic_array_tb.sv file

Waveform generated



- So the **first output doesn't match** the expected output victor in the **last two elements**, And the **second output doesn't match** the expected output in the **last element**, but the **third output** vector is **correct** and also the **last (fourth) output** vector is **correct**
- **However,** if we noticed the values of the accumulator at the **seventh clock** we will find that **all rows match the expected output vectors!**
- **So it is a problem of timing not functionality**
- In 4 x 4 inputs the output vector should be **delayed two clock cycles than expected** to get a correct output vectors
- If we could output **the whole matrix at once** we would've generate the right output after **seven cycles** only! (faster than the last output vector which should take eight cycles)
- I thought of delaying the output to be matched but I felt it would be impractical

Transcript

```
# output matrix rows are 0,  0,  0, 0
# output matrix rows are 80,  70,  52, 51
# output matrix rows are 240,  214,  188, 154
# output matrix rows are 400,  358,  316, 274
# output matrix rows are 560,  502,  444, 386
# ====================================
# Expected Output Matrix C:
# C[1][1]=80   C[1][2]=70   C[1][3]=60   C[1][4]=50
# C[2][1]=240  C[2][2]=214  C[2][3]=188  C[2][4]=162
# C[3][1]=400  C[3][2]=358  C[3][3]=316  C[3][4]=274
# C[4][1]=560  C[4][2]=502  C[4][3]=444  C[4][4]=386
# ====================================
# accumulator first row is: 80,  70,  60, 50
# accumulator second row is: 240,  214,  188, 162
# accumulator third row is: 400,  358,  316, 274
# accumulator fourth row is: 560,  502,  444, 386
```

Summary:

- **Functionality: Correct** if we consider the accumulator, **not Correct** in the first two rows if we consider the output vectors
- **Timing**: delayed by two clock cycle

Limitations:

From the **verifying functionality** section we deduce the following:

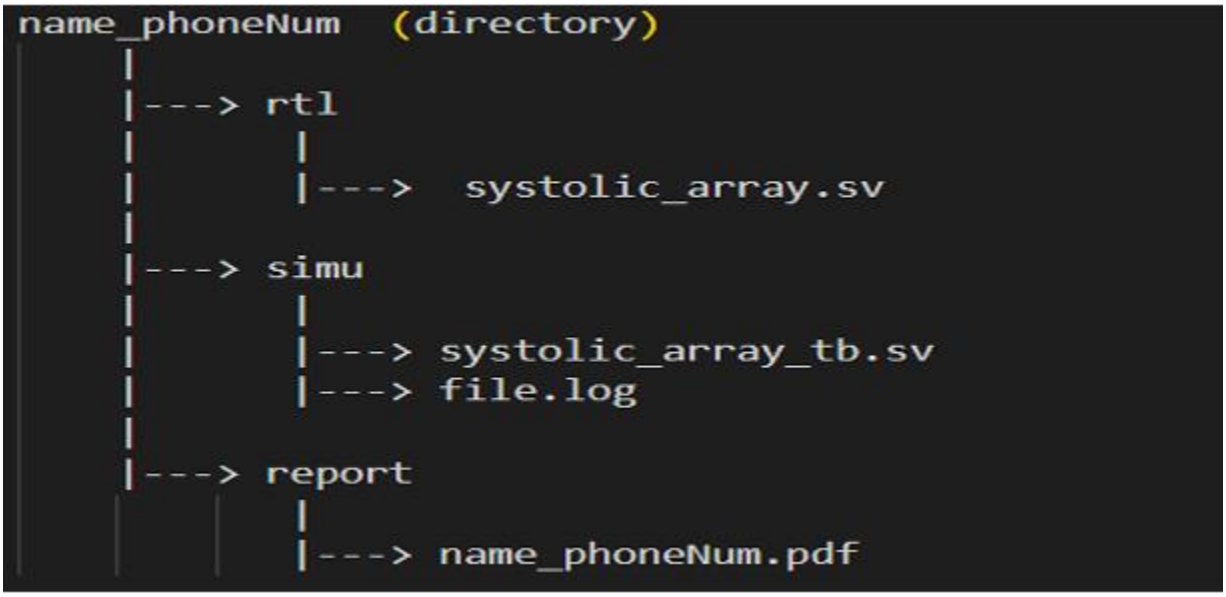
- The **functionality may be considered correct** in all cases **except** for the 4 by 4 matrix, but if we considered the **accumulator cells** we will find the functionality **always correct**
- The **timing** is getting **worse** every time we **increase the matrices order**, as the 2 by 2 matrix has the **perfect timing** but the 3 by 3 has a **one clock delay** and the 4 by 4 has a **2 clock delay** and so on…

Challenges

- ### Complex Modular Instantiation
  Initially considered using separate D flip-flops and processing elements, but managing the connections and functionality—especially for parameterized N—was too complex, so the idea was dropped.
- ### Buffering Full Matrices Violated Pipelining
  Tried storing entire input matrices and computing the result before pipelining the output, but this contradicted the core idea of streaming and pipelining inputs.
- ### Storing Pipelined Intermediate Inputs
  Faced difficulty storing intermediate pipelined values; although temp_a and temp_b captured initial inputs, handling delayed elements required additional pipelining logic.
- ### Output Timing Alignment
  Synchronizing outputs across varying pipeline stages was challenging, especially ensuring one row result per cycle without timing mismatches.
- ### Fully Parameterized Design Without Hardcoding
  Making the design scalable for any matrix size using parameters (e.g., N_SIZE) without hardcoding matrix dimensions added complexity in managing indexing and control flow.

GitHub Repository

I've created a GitHub repository that contains the same directory structure as the image illustrates:



You access the repository through this **link**

Note: I've added a separate .log file for every test, one for 2 x 2, another for 3 x 3 and another for 4 x 4.